APPLICATION

FOR

UNITED STATES LETTERS PATENT

APPENDIX C

TITLE: **PRONUNCIATION GENERATION IN SPEECH RECOGNITION**

APPLICANT: **JAMES K. BAKER, GREGORY J. GADBOIS, CHARLES E. INGOLD, STIJN A. VANEVEN AND JOEL PARK**

"EXPRESS MAIL" Mailing Label Number EM372130578US

Date of Deposit March 28, 1997

I hereby certify under 37 CFR 1.10 that this correspondence is being deposited with the United States Postal Service as **"Express Mail Post Office To Addressee"** with sufficient postage on the date indicated above and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

Lisa G. Gray

Lisa G. Gray

```
/*
   MODULE:  symphon.cpp

   PROJECT: DGNSRVR
   AUTHOR:  Stijn Van Even

   (C) Copyright Dragon Systems, Inc. 1997.

   ** DRAGON SYSTEMS CONFIDENTIAL **

-----------------------------------------------------------------------------

   DESCRIPTION:
       Module to generate pronunciations

-----------------------------------------------------------------------------

   TO DO LIST:

-----------------------------------------------------------------------------

   MODIFICATIONS:
     *tlib-revision-history*
    1 SYMPHON.CPP 04-Mar-97,19:03:10,`STIJN'
    2 SYMPHON.CPP 06-Mar-97,12:03:44,`STIJN'
    3 SYMPHON.CPP 06-Mar-97,20:26:22,`STIJN'
    4 SYMPHON.CPP 10-Mar-97,13:42:40,`STIJN' DDWIN Ver 2.52.084
    5 SYMPHON.CPP 13-Mar-97,15:17:32,`STIJN' DDWIN Ver 3.00.004
    6 SYMPHON.CPP 13-Mar-97,19:27:32,`STIJN' DDWIN Ver 3.00.010
    7 SYMPHON.CPP 17-Mar-97,15:19:50,`STIJN' DDWIN Ver 3.00.014
    8 SYMPHON.CPP 17-Mar-97,19:08:18,`ADAM' DDWIN Ver 3.00.019
    9 SYMPHON.CPP 18-Mar-97,10:11:36,`STIJN' DDWIN Ver 3.00.021
    10 SYMPHON.CPP 18-Mar-97,17:48:46,`STIJN' DDWIN Ver 3.00.024
    11 SYMPHON.CPP 19-Mar-97,17:50:32,`STIJN' DDWIN Ver 3.00.027.004 FR
    12 SYMPHON.CPP 19-Mar-97,19:16:36,`STIJN' DDWIN Ver 3.00.027.005 FR
    13 SYMPHON.CPP 20-Mar-97,19:30:02,`STIJN' DDWIN Ver 3.00.029
    14 SYMPHON.CPP 21-Mar-97,10:52:52,`STIJN' DDWIN Ver 3.00.029.001 FR
    15 SYMPHON.CPP 24-Mar-97,12:49:10,`STIJN' DDWIN Ver 3.00.031.002 FR
    16 SYMPHON.CPP 24-Mar-97,15:33:16,`ANNE' DDWIN Ver 3.00.031.001 ES
    17 SYMPHON.CPP 25-Mar-97,10:32:44,`STIJN' DDWIN Ver 3.00.035
    18 SYMPHON.CPP 25-Mar-97,15:11:16,`STIJN' DDWIN Ver 3.00.036
    19 SYMPHON.CPP 26-Mar-97,16:47:50,`STIJN' DDWIN Ver 3.00.039
    20 SYMPHON.CPP 27-Mar-97,12:26:26,`STIJN' DDWIN Ver 3.00.042
```

```
*tlib-revision-history*

 Revision 20 on Thu Mar 27 12:26:26 1997 by STIJN
DDWIN Ver 3.00.042
 maxNumHypos is set to 40 for Italian and 10 for Spanish

 Revision 19 on Wed Mar 26 16:47:51 1997 by STIJN
DDWIN Ver 3.00.039
 findSoundAlike had bug when when there was only one item in database
 that was a close match, and we landed just after it so that we never found
 a sound alike. Also put in a back-off so that now we should always get an
 assignment.

 Revision 18 on Tue Mar 25 15:11:16 1997 by STIJN
DDWIN Ver 3.00.036

 Revision 17 on Tue Mar 25 10:32:44 1997 by STIJN
DDWIN Ver 3.00.035
 Added some checks in AssignOnePron()

 Revision 15 on Mon Mar 24 12:49:10 1997 by STIJN
DDWIN Ver 3.00.031.002 FR
 Added robustness against long and bizarre spelling strings i.e "aslkdj ask as
 xxy as asddcds %"

 Revision 14 on Fri Mar 21 10:52:52 1997 by STIJN
DDWIN Ver 3.00.029.001 FR
 maxNumHypos is set to one when we hypothesize an acronym.

 Revision 13 on Thu Mar 20 19:30:02 1997 by STIJN
DDWIN Ver 3.00.029
 Added more pron-helper support. It is now applied in the server.
 We now create a new pron for (1) a new word; (2) an existing word with a
 zero based model and delete that model when the guesser is successful;
 (3) an existing word with a pron that was misrecognized. Additional prons
 are added up to 3 prons. After that the 3rd pron is being replaced each time
 a correction is done on that word.
 We now create one pronunciation even when there is no utt. We take the top
 hypothesis and assign it to the word. If it is a bad pron, the error correction
 will recover it with a better one.

 Revision 12 on Wed Mar 19 19:16:35 1997 by STIJN
DDWIN Ver 3.00.027.005 FR
```

```
 Revision 11 on Wed Mar 19 17:50:33 1997 by STIJN
DDWIN Ver 3.00.027.004 FR

 Revision 10 on Tue Mar 18 17:48:46 1997 by STIJN
DDWIN Ver 3.00.024
 Changed char* into char far * for the file name arrays.
 Added more support to add pronunciation helper models.

 Revision 9 on Tue Mar 18 10:11:36 1997 by STIJN
DDWIN Ver 3.00.021
 Array of prefilter names did not seperate new language names with commas.

 Revision 8 on Mon Mar 17 19:08:19 1997 by ADAM
DDWIN Ver 3.00.019
 Fixed bug where word with sound-alike pron wasn't getting created

 Revision 7 on Mon Mar 17 15:19:50 1997 by STIJN
DDWIN Ver 3.00.014
 Removed some memory leaks due to "new" allocations in initialize().
 Now that initialize can be called several times, this was a bug.

 Revision 6 on Thu Mar 13 19:27:31 1997 by STIJN
DDWIN Ver 3.00.010
 We now always hypothezise an initials pron for each word.

 The module now checks whether the pron guessing data files
 exist, before trying to load them. If they do not exist,
 then we do not initialize the guesser.

 Revision 5 on Thu Mar 13 15:17:31 1997 by STIJN
DDWIN Ver 3.00.004
 We now delete words from hSymphonVoc. Before, we only deleted from the
 temp state, which caused an sdapi WORD ALREADY EXISTS error if the same
 word was pron guessed again.
 Added support to add additional prons for an existing word. Currently not
 done by the caller, due to a driver bug that can not handle adding
 additional prons to an ALPHA_ORDERED state
 The define EXISTFILE is not defined. It will be removed once I have
 made sure that checking the existence of the pron files works correctly.


 Revision 4 on Mon Mar 10 13:42:39 1997 by STIJN
DDWIN Ver 2.52.084
*/
```

```
///////////////////////////////////////////////////////////////////////////

#include "symphon.h"

#ifdef STRESSADDED
#define AE_NUMTOPHYPOS  200
#else
#define AE_NUMTOPHYPOS  300
#endif

#define GER_NUMTOPHYPOS 200
#define FRA_NUMTOPHYPOS  10
#define ITA_NUMTOPHYPOS  40
#define ESP_NUMTOPHYPOS  10

#define NUMTOPHYPOS     700
#define SYMPHONMAXPRON  200


// static globals
Symbol** SymbolStatistics::s2pTable;
Phone**  SymbolStatistics::p2sTable;
char*    SymbolStatistics::name = "Pronunciation-engine";


static Symbol** symbolChart;
static int      numNewProns = 0;
static int      numAllNewProns = 0; // used to keep track of number of
                                                              // new IDS in temp.voc so that a decision
                                                              // to close and reopen voc can be made

static int      numNewPronsWithStress = 0;
static SD_VOC   hSymphonVoc = 0;
static int      *scoreArray = NULL;
static char     *realSpelling = NULL;

#ifdef DOBIGRAMS
BOCL doPhonemeDigrams = FALSE;
#endif


// place where we store the hypoScores
SD_WORD     idArray         [NUMSELECTED];
int     distanceArray   [NUMSELECTED];
```

-4-

```
struct  addWord newWord [NUMSELECTED];


//---------------------------------------------------------------------------
//
int recogCompareHypothesis( const void* given, const void* test )
{
    PronIdHypo* ph1 = (PronIdHypo* )test;
    PronIdHypo* ph2 = (PronIdHypo* )given;

    int rtn = ph1->score - ph2->score;

    if( rtn == 0 )
        rtn = ph1->wordId - ph2->wordId;

    return rtn;
}


//-----------------------------------------------------------------------
//
int trainCompareHypothesis( const void* given, const void* test )
{
    PronHypo* ph1 = (PronHypo* )test;
    PronHypo* ph2 = (PronHypo* )given;

    int rtn = ph1->score - ph2->score;

    if( rtn == 0 )
        rtn = strlen( ph1->spellingData ) - strlen( ph2->spellingData );

    return rtn;
}


//---------------------------------------------------------------------------
int propagateCompareHypothesis( const void* given, const void* test )
{
    PronHypo* ph1 = (PronHypo* )test;
    PronHypo* ph2 = (PronHypo* )given;

    int rtn;
    if( ph1->ithSymbol <= 0 && ph2->ithSymbol <= 0 )
        rtn = ph1->score - ph2->score;
    else if( ph1->ithSymbol <= 0 && ph2->ithSymbol > 0 )
        rtn = ph1->score - ( ph2->score / ph2->ithSymbol );
    else if( ph1->ithSymbol > 0 && ph2->ithSymbol <= 0 )
        rtn = ( ph1->score / ph1->ithSymbol ) -  ph2->score;
    else
        rtn = ( ph1->score / ph1->ithSymbol ) -
    ( ph2->score / ph2->ithSymbol );

    if( rtn == 0 )
        rtn = strlen( ph1->spellingData ) - strlen( ph2->spellingData );

    return rtn;
}


//----------------------------------------------------------------------------
//
void PronHypo::propagate( PronHypoPQ* pq )
{
    for( Symbol* sym = SymbolStatistics::findFirstSymbol( spellingData );
         sym;
         sym = SymbolStatistics::findNextSymbol( spellingData, sym ) )
    {
        int symsplen = strlen( sym->spellingString );

        if( strncmp( sym->spellingString, spellingData, symsplen ) )
            continue;

        for( Phone* pho = (Phone*) sym->phone; pho; pho = (Phone*)pho->next() )
        {
            char *pr = pho->pronunciationString;

            PronHypo* hypo = new
            PronHypo( pr, sym, pronunciationData + strlen( pr ),
                      0, nSymbols, numOutputPhonemes,
                      spellingData + symsplen, symStats, this );

            hypo->score = score + pho->penalty;
            hypo->numOutputPhonemes += pho->pronLength;

            pq->push( hypo );
        }
    }
}


//-----------------------------------------------------------------------------
void convertStressedSchwa( char &lastPhonemeOfPrev, char &firstPhonemeOfNext )
{
    if( lastPhonemeOfPrev == 'u' )
        lastPhonemeOfPrev = '@';
    if( firstPhonemeOfNext == 'u' )
        firstPhonemeOfNext = '@';
}



//-----------------------------------------------------------------------------
// take longest symbol during hypothesizing
int PronHypo::propagateLongest( PronHypoPQ* pq )
{
    Symbol* maxSym = (Symbol*) symbolChart[ ithSymbol ];

    ithSymbol++;  // counter of which symbol we are working with

    // to avoid GP fault in case maxSym == NULL
    if( !maxSym ) return 0;

    for( int i = 0; i < maxSym->nPhones; i++ )
    {
        Phone* pho = maxSym->phoneArray[i];

        char* pr = pho->pronunciationString;

        PronHypo* hypo = new PronHypo( pr, maxSym, pronunciationData, ithSymbol,
                nSymbols, numOutputPhonemes, spellingData, symStats, this );

        // adjust member data of hypo
        hypo->numOutputPhonemes += pho->pronLength;
        hypo->pronLength = pho->pronLength;
        hypo->score = score + pho->penalty;

#ifdef DOBIGRAMS
        if( doPhonemeDigrams )
        {
            // ' ' stands for silence at beginning or end of word
            char lastPhonemeOfThis = ' ';
            if ( pronunciationString )
                lastPhonemeOfThis = pronunciationString[ pronLength-1 ];
            else
                lastPhonemeOfThis = ' ';

            // the following var is never used.
            char firstPhonemeMaxSym = pho->pronunciationString[0];

            // the phoneme pair table does not have stressed schwas,
            // some rules do
            convertStressedSchwa( lastPhonemeOfThis, firstPhonemeMaxSym );

            int diphScore = 0;

            if( lastPhonemeOfThis != '0' && firstPhonemeMaxSym != '0' )
                diphScore = LS->diphoneLog( lastPhonemeOfThis,
                                            firstPhonemeMaxSym );


            hypo->score += diphScore;
        }
#endif
        pq->push( hypo );

    }

    return 1;
}


//----------------------------------------------------------------------
//    SymbolStatistics
//----------------------------------------------------------------------
char* parseToNextSpace( unsigned char* sin )
{
    while( *sin != '\0' && *sin != '\r' && *sin != '\n' )
    {
        if( *sin == ' ' )
        {
            *sin++ = '\0';

            return (char*) sin;
        }
        else
            ++sin;
    }
    *sin = '\0';

    return 0;
}

//----------------------------------------------------------------------
// expand to preprocessing step
void SymbolStatistics::preProcessSpelling( const char* sin, char* sout )
{
    unsigned char unaccentedString[WORDLENGTH];

    // map extended characters
    letterEditor->mapExtLettersToLetters( (unsigned char*) sin,
                                          unaccentedString,
                                          WORDLENGTH );

    // take care of some idiosyncracies and character delimiters
    letterEditor->prepareString( unaccentedString, sout, WORDLENGTH );
}


// ----------------------------------------------------------------------
// fill array to access the phones more quickly
void SymbolStatistics::fillPhoneArray()
{
    for( int  i= 0; i < 256; i++ )
        if( s2pTable[i] != (Symbol*) NULL )
        {
            for( Symbol* sym = (Symbol*) s2pTable[i]; sym;
                 sym = (Symbol*) sym->next() )
            {
                sym->phoneArray = new Phone* [ sym->nPhones ];

                int k = 0;
                for( Phone* pho = sym->phone; pho && k < sym->nPhones;
                     pho = (Phone*) pho->next() )
                {
                    sym->phoneArray[ k ] = pho;
                    k++;
                }
```

```
// ---------------------------------------------------------------------------
//
BOOL SymbolStatistics::fileExists( LPCSTR fileName )
{
    int ffFlag = 0;
    struct ffblk ffBlk;

    if (0 == findfirst(fileName, &ffBlk, ffFlag) )
    {
#ifdef DEBUG
        char str[256];
        wsprintf( str, "File is %s\n", fileName );
        OutputDebugString( str );
#endif
        return TRUE;
    }

    return FALSE;
}


// ---------------------------------------------------------------------------
// fill array to access the phones more quickly
int SymbolStatistics::loadPrnRules( LPCSTR prnRulesFileName )
{
    unsigned char spelling[WORDLENGTH];
    unsigned char pronunciation[WORDLENGTH];
//#ifdef EXISTFILE
    if( !fileExists( prnRulesFileName ) )
        return 0;
//#endif
    // save away old value of "multiple-ids"
    SD_PAR hParMultipleIds = sdapi.SDPar_GetHandle("multiple-Ids");
    BOOL saveMultipleIDs;
    sdapi.SDPar_GetValue( hParMultipleIds, &saveMultipleIDs,
                          sizeof( saveMultipleIDs) );
    // set multiple IDs
    BOOL bMultipleIds = TRUE;
    sdapi.SDPar_SetValue( hParMultipleIds, &bMultipleIds, sizeof(bMultipleIds) );

    SD_VOC hPrnRuleVoc;

    if( ( hPrnRuleVoc = sdapi.SDVoc_Open( prnRulesFileName,"rw" ) ) == 0 )
```

```
        return 0;

    sdapi.SDWord_Load (hPrnRuleVoc, 0 /* all ids */);

    SD_WORD hWord;
    SD_WORD_ITERATOR wordIt;
    sdapi.SDWord_Iterate( hPrnRuleVoc, &wordIt );

    while( ( hWord = sdapi.SDWord_Next( &wordIt ) ) != 0 )
    {
        SD_WORD idBuf[SYMPHONMAXPRON]; // should be enough
        int nAlts = sdapi.SDWord_ListIds( hPrnRuleVoc, hWord, idBuf, SYMPHONMAXPRON );

        sdapi.SDWord_GetName( hPrnRuleVoc, hWord, (char*) spelling,
                              sizeof( spelling ) );

        for( int i = 0; i < nAlts && i < SYMPHONMAXPRON; i++ )
        {
            sdapi.SDWord_GetPronunciation( hPrnRuleVoc, idBuf[i], pronunciation,
                                           sizeof( pronunciation ) );


            // assign LM count
            int logScore = sdapi.SDWord_GetLm1Count( hPrnRuleVoc, idBuf[i] );

            addToTablesIfNecessary( spelling, pronunciation, logScore );
        }
    }

    sdapi.SDPar_SetValue( hParMultipleIds, &saveMultipleIDs,
                          sizeof( saveMultipleIDs ) );

    sdapi.SDVoc_Close (hPrnRuleVoc);

    fillPhoneArray();

    return 1;
}


//----------------------------------------------------------------------------
//
int SymbolStatistics::loadPreFile( LPCSTR prefFileName )
{
```

```
//#ifdef EXISTFILE
    if( !fileExists( prefFileName ) )
        return 0;
//#endif

    // set prefilter voc first
    if( ( hVocSoundAlike = sdapi.SDVoc_Open( prefFileName,"rw" ) ) == 0 )
        return 0;

    sdapi.SDWord_Load( hVocSoundAlike, 0 );

    return 1;
}

//---------------------------------------------------------------------------
//
void SymbolStatistics::setLangVariable(LPSTR curLang, SD_VOC hVoc)
{
    sdapi.SDVoc_SetEnv (hVoc, "DragonLanguage", curLang, strlen(curLang)+1 );
    sdapi.SDVoc_SetEnv (hVoc, "_1",             curLang, strlen(curLang)+1 );
}


//---------------------------------------------------------------------------
// set up a recognizer class that has a channel, a user and a voc
// There will be no event handler, since the utt will be given through the
// application that call generate( spelling, hUtt );
int SymbolStatistics::initialize( SD_USER hUser, int language, LPCSTR rulesData,
                                  LPCSTR preData)
{
    scoreArray = new int[1000];

    lang = language;

    hPronUtt = 0;

    assert ( language <= LANG_CT );

    SD_PAR hParMultipleIds = sdapi.SDPar_GetHandle("multiple-Ids");
    BOOL saveMultipleIDs;
    sdapi.SDPar_GetValue( hParMultipleIds, &saveMultipleIDs,
                          sizeof( saveMultipleIDs) );
    BOOL bMultipleIds = TRUE;
    sdapi.SDPar_SetValue( hParMultipleIds, &bMultipleIds, sizeof(bMultipleIds) );
```

```
// load prefiltering info
if( loadPreFile( preData ) == 0 )
    return 0;

if( loadPrnRules( rulesData ) == 0 )
    return 0;

user = hUser;

// reset multiple IDs
sdapi.SDPar_SetValue( hParMultipleIds, &saveMultipleIDs,
                      sizeof(saveMultipleIDs) );

// create temp vocabulary and set name
hSymphonVoc = sdapi.SDVoc_New();
if( hSymphonVoc == 0 ) return 0;
sdapi.SDVoc_SetFileName (hSymphonVoc, "temp.voc");

// set language variable
switch ( language ) {
 case ENU:
    setLangVariable( "UsEnglish", hSymphonVoc );
    break;

 case ENE:
    setLangVariable( "BritishEnglish", hSymphonVoc );
    break;

 case GER:
    setLangVariable( "German", hSymphonVoc );
    break;

 case ESP:
    setLangVariable( "Spanish", hSymphonVoc );
    break;

 case FRA:
    setLangVariable( "French", hSymphonVoc );
    break;

 case ITA:
    setLangVariable( "Italian", hSymphonVoc );
    break;
    case NIH:
       setLangVariable( "Japanese", hSymphonVoc );
       break;

    default:
       return 0;
   }

   return 1; // return 1 if  vocs and user are set successfully
}


//--------------------------------------------------------------------------------
//
void SymbolStatistics::setDefaultMaxNumHypos()
{
    if( hPronUtt == 0 )
        maxNumHypos = 1;

    else
    {
        switch ( lang ) {
         case ENU:
         case ENE:
         case NIH:
            maxNumHypos = AE_NUMTOPHYPOS;
            break;

         case GER:
            maxNumHypos = GER_NUMTOPHYPOS;
            break;

         case FRA:
            maxNumHypos = FRA_NUMTOPHYPOS;
            break;

         case ESP:
            maxNumHypos = ESP_NUMTOPHYPOS;
            break;

         case ITA:
            maxNumHypos = ITA_NUMTOPHYPOS;
            break;
    // given a spelling generate Prons
    if( generateProns( spelling, description, vocName, isInitials, mapStrategy ) > 0 )
    {
        if( hUtt )
            return selectBestProns( hUtt, vocName, targetStateName, doLmScore );
        else
            return assignOnePron();
    }

    return 0;
}


//--------------------------------------------------------------------------
// generate pronunciations in temp state
// returns number of prons
int SymbolStatistics::generateProns( LPCSTR spelling, LPCSTR description,
                                     LPCSTR vocName, int isInitials,
                                     int mapStrategy )
{

    PronHypoPQ* topPQ = new PronHypoPQ( propagateCompareHypothesis,
                                        NUMTOPHYPOS );

    vocName; // beat a warning

    char newSpelling[WORDLENGTH];
    unsigned char pronunciation[WORDLENGTH];

    int nHypo = 0;

    // realSpelling is a static
    strcpy( realSpelling, spelling );

    // creates a temp state or cleans it out
    // when we learn a language we want to keep hypo's from othertongue.
    // The subSequent call will put additional stuff in them
    initializeState();

    makeCap( description, newSpelling );

    if( isInitials )
    {
        maxNumHypos = 1;
        default:
            maxNumHypos = AE_NUMTOPHYPOS;
        }
    }
}


//-------------------------------------------------------------------------------
//
SD_WORD SymbolStatistics::generate( LPCSTR spelling,     // real spelling
                                    LPCSTR description, // could be sound alike
                                    SD_UTT hUtt,         // handle of utterance
                                    LPCSTR vocName,      // vocabulary
                                    ~~LPCSTR targetStateName, // state name~~
                                    int doLmScore,       // add lm score to total
                                    BOOL isInitials,     // U.F.O.
                                    int mapStrategy )    // map phoneme to
                                                         // univeral set
{
    if( hUtt )
    {
        SD_UTT_INFO sdUttInfo;
        sdapi.SDUtt_GetInfo( hUtt, &sdUttInfo );
        if ( sdUttInfo.rejCode != 0 )
        {
            return 0; // utterance is invalid
        }
    }

    // let us not do this twice: same utt on the same spelling
    if( hPronUtt == hUtt && hUtt != 0 &&
        !strcmp( description, guessedWord ) )
        return 0;


    strncpy( guessedWord, description, WORDLENGTH );
    guessedWord[ strlen( description ) ] = '\0';

    hPronUtt = hUtt;

    // this can be 0
    setDefaultMaxNumHypos();
```

```
        spellingDecoder( topPQ, newSpelling,  NULL, 0 );

        nHypo = topPQ->count();

        for( int i = nHypo-1; i >= 0; i-- )
        {
            PronHypo* ph = topPQ->pop();

            if( ph == NULL )
                // would be bizarre, but you never know
                continue;

            // extract pronunciation
            getPronunciation( ph, pronunciation, isInitials ):

            // adds pron to state and generates stress levels
            addPronToState( realSpelling, pronunciation,
                            ph->score, mapStrategy );
        }

    }

    else
    {
        if ( maxNumHypos > 1 )
        {
            // only in case when we have determined to make more than one hypo
            // generate an initials-pron

            maxNumHypos = 1;

            makeCap( description, newSpelling );

            spellingDecoder( topPQ, newSpelling,  NULL, 0 );

            // default values defined per language
            setDefaultMaxNumHypos();
        }

        // now generate prons for lower-cased version
        preProcessSpelling( description, newSpelling );

        spellingDecoder( topPQ, newSpelling,  NULL, 0 );
        nHypo = topPQ->count();

        for( int i = nHypo-1; i >= 0; i-- )
        {
            PronHypo* ph = topPQ->pop();

            if( ph == NULL )
                // would be bizarre, but you never know
                continue;

            // extract pronunciation
            getPronunciation( ph, pronunciation, isInitials );

            // adds pron to state and generates stress levels
            addPronToState( realSpelling, pronunciation,
                            ph->score, mapStrategy );

        }
    }

    delete topPQ;

    return nHypo;
}

//---------------------------------------------------------------------------
//
SD_WORD* SymbolStatistics::getChoices( )
{
    return idArray;
}


//---------------------------------------------------------------------------
//
int SymbolStatistics::getConfidence()
{
    return 50;
}


//---------------------------------------------------------------------------
```

```
//
int* SymbolStatistics::getDistances()
{
    return distanceArray;
}


//---------------------------------------------------------------------------
// Use utterance to select best pronunciation
SD_WORD SymbolStatistics::assignOnePron()
{
    char wordName[WORDLENGTH];
    unsigned char pron[WORDLENGTH];

    wsprintf( wordName, "%s__1", realSpelling );
    SD_WORD hWord = sdapi.SDWord_GetHandle( hSymphonVoc, wordName );

    if( !hWord )
        // do not continue, the word was not added to the voc for some reason.
        // probably, due to an illegal phoneme
        return 0;

    sdapi.SDWord_GetPronunciation( hSymphonVoc, hWord,
                                   (unsigned char*) pron, WORDLENGTH );

    SD_WORD newId = sdapi.SDWord_New( hSymphonVoc, realSpelling );

    sdapi.SDWord_SetPronunciationWithType( hSymphonVoc, newId,
                                           (unsigned char*) pron,
                                           SD_PRONTYPE_GENERAL );
    // give an lm count of 1
    sdapi.SDWord_SetLm1Count( hSymphonVoc, newId, 1 );

    distanceArray[0] = newId;

    return newId;
}


//---------------------------------------------------------------------------
// Use utterance to select best pronunciation
SD_WORD SymbolStatistics::selectBestProns( SD_UTT hUtt, LPCSTR vocName,
                                           LPCSTR stateName, int doLmScore )
{
```

```
char pron[WORDLENGTH];
 char buf[WORDLENGTH];

 vocName;
 stateName;

 SD_STATE hState = 0;
 SD_WORD hWord = 0;
 RECOG_STATUS gRecogStatus;   // status of last recog
 RECOG_RESULT_ENTRY  gRecogResults[ NUMSELECTED ]; // choice list structures

 if( realSpelling[0] == '\0' ) return 1;

 PronIdHypoPQ hypoPQ( recogCompareHypothesis, 1000 );

 int n = 0; // used for number of PQ hypo's

 hState = sdapi.SDState_GetHandle( hSymphonVoc, "temp", 0 );

 int gnEntries = 0;

 // recognize
 if( ( gnEntries = sdapi.SDState_Recog1( hSymphonVoc,
                                         hState,
                                         hUtt,
                                         0,
                                         0,
                                         gRecogResults,
                                         sizeof( gRecogResults ),
                                         &gRecogStatus ) ) != 0 )
 (
     gnEntries;

     for( int i = 0; i < gRecogStatus.nChoices && i < NUMSELECTED; i++ )
     (
         sdapi.SDWord_GetName( hSymphonVoc,
                               gRecogResults[i].wordSpec[0].hWord, buf,
                               WORDLENGTH );

         SD_WORD tempId = gRecogResults[i].wordSpec[0].hWord;

         // distance
         int distance = gRecogResults[i].distance;
```

```
        // new Spelling has ck->K, x->xX, no punctuation and no digits
        PronIdHypo* hypo = new PronIdHypo( tempId, 0 );

        // add the score
        if( doLmScore )
            hypo->score = ( scoreArray[tempId%1000] / LMFACTOR ) + distance;
        else
            hypo->score = distance;

        // get pron
        sdapi.SDWord_GetPronunciation( hSymphonVoc, tempId,
                                       (unsigned char* ) pron, WORDLENGTH );

        hypoPQ.push( hypo );
    }

    // save NUMSELECTED words in newWord structure; now NUMSELECTED == 1
    n = hypoPQ.count();

    for( i = 0; i < n && i < NUMSELECTED; i++ )
    {
        hWord = hypoPQ[n-(i+1)]->wordId;

        sdapi.SDWord_GetName( hSymphonVoc, hWord, newWord[i].spelling,
                              WORDLENGTH  );

        sdapi.SDWord_GetPronunciation( hSymphonVoc, hWord,
                                       (unsigned char*) pron, WORDLENGTH );

        strcpy( (char *)newWord[i].pron,(char *) pron );
        distanceArray[i] = gRecogResults[i].distance;
    }


    // remove IDs from temp state
    initializeState();
}
// recognition rejected
else
    return 0;

// declare for readability
SD_WORD newId = 0;
```

```
    for( int i = 0; i < n && i < NUMSELECTED; i++ )
    {
        // Add word to vocabulary
        strcpy( newWord[i].spelling, realSpelling );

        newId = sdapi.SDWord_New( hSymphonVoc, newWord[i].spelling );

        sdapi.SDWord_SetPronunciationWithType( hSymphonVoc, newId,
                                              (unsigned char*)newWord[i].pron,
                                              SD_PRONTYPE_GENERAL );
        // give an lm count of 1
        sdapi.SDWord_SetLmlCount( hSymphonVoc, newId, 1 );

        idArray[ i ] = newId;
    }

    hypoPQ.removeAll();

    // return top choice. If NUMSELECTED is larger than one, we keep the choice
    // list in this structure.
    return idArray[0];
}

#define NALTSPRON 400


// -----------------------------------------------------------------------------
// Compute prefiltering
SD_WORD SymbolStatistics::findSoundAlike( SD_VOC hVocSoundAlike,
                                           unsigned char* pPron,
                                           BOOL IsAltPron )
{
#define PREFILTWORDS 200

    SD_WORD hSoundAlikeWord = 0;

    SD_WORD prefiltArray[PREFILTWORDS];
    unsigned char SoundAlikePron[WORDLENGTH];
    BOOL bFoundPron = FALSE;
    uns32 index = 0;
    int nWords = 0;

#if DEBUG
```

```
    FILE* preFile = fopen( "prelog", "a+w");
#endif

    // try to find closest match
    index = sdapi.SDWord_Lookup( hVocSoundAlike, (char*)pPron );

    if( index > 1 )
        index = index - 1; // get the preceding word. Index is the postion
                           // between the previous closest and the next closest
                           // match. Sometimes the previous one is a valid partial
                           // match, but it is the only one in the database, since
                           // some of these phoneme sequences are rare. Not decreasing
                           // by one would lead to a miss


    nWords = sdapi.SDWord_List( hVocSoundAlike, index,
                                prefiltArray, PREFILTWORDS );

    // no prefilter assignement
    if( nWords == 0 ) return 0;

    // make comparisons
    int pronLength = 0;
    unsigned char pronBuf[WORDLENGTH];

    // pronunciation of new word
    if( ( pronLength = strlen( (char*)pPron ) ) > 4 )
        pronLength = 4;

    // should not be longer than 4 phonemes
    // check all the variant pronunciations in decreasing length
    // until match is found
    while( pronLength >= 1 )
    {
        if( !IsAltPron && pronLength == 1 )
            break;

        for( int i = 1; i < nWords && i < NALTSPRON; i++ )
        {
            pronBuf[0] = '\0';

            sdapi.SDWord_GetName( hVocSoundAlike, prefiltArray[i],
                                  (char*) pronBuf, sizeof( pronBuf ) );
            // for safety
            int saveLength = pronLength;

            if( pronLength > strlen( (char*) pronBuf ) )
                pronLength = strlen( (char*) pronBuf );

            if( strncmp( ( const char* ) pPron,
                         (const char*) pronBuf, pronLength ) == 0 )
            {
                hSoundAlikeWord = prefiltArray[i];
                bFoundPron = TRUE;
                break;
            }

            if( bFoundPron ) break;

            pronLength = saveLength;

        }

        if( bFoundPron ) break;

        // there is only one choice
        if( nWords == 1 )
        {
            hSoundAlikeWord = prefiltArray[nWords-1];

            bFoundPron = TRUE;
            break;
        }

        pronLength--;
    }

    if( hSoundAlikeWord == 0 )
        // in case nothing was found, do something
        if( nWords )
            // make sure the prefiltArray is not empty
            hSoundAlikeWord = prefiltArray[0];


#ifdef DEBUG
    // for now take the first word
    if( hSoundAlikeWord )
    {
        sdapi.SDWord_GetName( hVocSoundAlike, hSoundAlikeWord,
                              (char*) SoundAlikePron,
                              sizeof( SoundAlikePron ) );

        // just to check
        phonemeSet->convertPronToAruba( SoundAlikePron );

        fprintf( preFile, "guessed == %s; prefilter == %s\n", pPron, SoundAlikePron );
    }

    fclose( preFile );
#endif


    return hSoundAlikeWord;
}



// ---------------------------------------------------------------------------
// adds word from hSymphonVoc into hVoc
SD_WORD SymbolStatistics::addWordWithWsa( LPCSTR pWord, SD_VOC hVoc,
                                          SD_STATE hState, long uniCount )
{
#ifdef DEBUG
    logFile = fopen( "good.ok", "w+a" );
    assert( logFile );
#endif

    unsigned char pPron[WORDLENGTH];
    unsigned char altPron[WORDLENGTH];
    unsigned char origPron[WORDLENGTH];
    unsigned char dupPronBuf[WORDLENGTH];

    BOOL bHasNoPron = FALSE;
    BOOL pronWithUtt = FALSE;

    SD_WORD dupIdBuf[NALTSPRON];  // should be enough
    int nDupAlts = 0;

    SD_WORD hTempWord = sdapi.SDWord_GetHandle( hSymphonVoc, pWord );
    if( !hTempWord ) return 0;

    sdapi.SDWord_GetPronunciation( hSymphonVoc, hTempWord, pPron,
                                    sizeof( pPron ) );
assert( hVoc );

// --- save the pPron
strcpy( (char*)origPron, (char*)pPron );

pPron[4] = '\0';

// --- this will take care of the high bit syllable boundary stuff
phonemeSet->convertPronToAruba( pPron );

// --- make an stressed or unstressed pron from pPron
phonemeSet->addStressOrUnstress( pPron, altPron );

// --------- Duplicate checking ---------------------------
// First check for duplicates, since driver allows multiple instances
// of the same word and prons for unknown theoretical reason
// set multiple IDS so that we can access ids in pre-voc
SD_PAR hParMultipleIds = sdapi.SDPar_GetHandle("multiple-Ids");
BOOL saveMultipleIDs;
sdapi.SDPar_GetValue( hParMultipleIds, &saveMultipleIDs,
                sizeof( saveMultipleIDs) );
BOOL bMultipleIds = TRUE;
sdapi.SDPar_SetValue( hParMultipleIds, &bMultipleIds, sizeof(bMultipleIds) );

SD_WORD w = sdapi.SDWord_GetHandle( hVoc, pWord );

// ------------ check if pronunciation already exists -------------------
if( w )
{
    // word exists...
    nDupAlts = sdapi.SDWord_ListIds( hVoc, w, dupIdBuf, NALTSPRON );

    if( nDupAlts == 0 )
        bHasNoPron = TRUE;

    for( int i = 0; i < NALTSPRON && i < nDupAlts; i++ )
    {
        sdapi.SDWord_GetPronunciation( hVoc, dupIdBuf[i],
                                       dupPronBuf, sizeof( dupPronBuf ) );

        if( dupPronBuf(0) == '\0' )
            bHasNoPron = TRUE;
            phonemeSet->convertPronToAruba( dupPronBuf );

            if( strcmp( ( const char* ) dupPronBuf,
                        (const char*) origPron ) == 0 )
            {
                // they are the same, so we already have this word/pron
                // in the vocabulary
                sdapi.SDPar_SetValue( hParMultipleIds, &saveMultipleIDs,
                                      sizeof(saveMultipleIDs) );
                return dupIdBuf[i];
            }
        }
} // end of duplicate checking


// ------------------ query property on word -----------------------------
if( w )
{
    // get pronWithUtt value
    SD_STATE_WORD_INFO1 stateInfo;

    sdapi.SDState_GetWordInfo1( hVoc, hState, w, &stateInfo );

    if( stateInfo.isInState )
        sdapi.SDState_GetWordEnv( hVoc, hState, w, "pronWithUtt",
                                  &pronWithUtt, sizeof( BOOL) );
}

SD_WORD hNewWord = 0;

// ---------------- if there is a sound alike voc -----------------------
if( hVocSoundAlike )
{
    SD_WORD hSoundAlikeWord;

    hSoundAlikeWord = sdapi.SDWord_GetHandle( hVocSoundAlike, (char*)pPron );

    if( hSoundAlikeWord == 0 )
        // try to find with stressed version
        hSoundAlikeWord = sdapi.SDWord_GetHandle( hVocSoundAlike,
                                                  (char*)altPron );

    if( hSoundAlikeWord == 0 )
    {
```

```
    hSoundAlikeWord = findSoundAlike( hVocSoundAlike, pPron, 0 );

    if( hSoundAlikeWord == 0 )
        hSoundAlikeWord = findSoundAlike( hVocSoundAlike, altPron, 1 );
}

// ----------------------- assign pron and LM1 count -----------------
if( hSoundAlikeWord )
{

    // If a pron exist for a word, we will never get here. In
    // future it is a possibility to generate pronunciation
    // helper models. We then need to bring the proper mechanism
    // to add the word to the voc and state through the server
    // calls. Now this is done in API_STAT.CPP after the pron
    // guessing.

    if( w && bHasNoPron )
    {
        // word has been already added, but did not have a pron
        // model.
        //
        // (1) it is the case when words are added via
        // DDX, and now they get pronunciation through the training
        // dialog, or through tracking
        //
        // (2) the word has a zero based model
        //
        hNewWord = w;
    }
    else
        // (1) The word did not exist
        // (2) The word existed with a pron, different from the newly
        //     found pron.
        // (3) The word existed with a pron, but it was generated without
        //     an Utterance: therefore replace it.

        if( nDupAlts == 1 && pronWithUtt == FALSE )
        {
            hNewWord = w;
            if( hPronUtt )
            {
                // if there is an utterance, trample previous one
                // and env to TRUE
```

```
                    BOOL bPronWithUtt = TRUE;
                    sdapi.SDState_SetWordEnv( hVoc, hState, w, "pronWithUtt",
                                              &bPronWithUtt, sizeof( BOOL) );
                }
            }
            else if( nDupAlts < MAXPRON )
            {
                hNewWord = sdapi.SDWord_New( hVoc, pWord );
            }
            else
                // the word does exist and has MAXPRON pronunciations:
                // We keep overwriting the last pron of the series if
                // max allowable prons are there.
                hNewWord = dupIdBuf[MAXPRON-1];

        // set its pronunciation, usable in all situations. If you want
        // it only used in CSR and phrase-building, use
        // SD_PRONTYPE_CSRONLY
        sdapi.SDWord_SetPronunciationWithType( hVoc, hNewWord, origPron,
                                               SD_PRONTYPE_GENERAL );

#ifdef DEBUG
        SD_WORD_INFO info;
        sdapi.SDWord_GetInfo( hVoc, hNewWord, &info );

        if( info.hasModel )
        {
            char debugStr[WORDLENGTH];
            wsprintf( debugStr, "%s %s\n", pWord, origPron );
            OutputDebugString( debugStr );
        }
#endif
        // assign LM count
        sdapi.SDWord_SetLm1Count( hVoc, hNewWord, uniCount );

        // assign prefiltering
        sdapi.SDWord_SetWordStartSoundAlike(hVoc, hNewWord, hVocSoundAlike,
                                            hSoundAlikeWord );
    }

#ifdef DEBUG
    else
    {
        char debugStr[WORDLENGTH];
```

```
            wsprintf( debugStr, "No prefilter model found for word: %s, pron: %s\n", pWord, origPron
);          OutputDebugString( debugStr );
        }
#endif

    }

    sdapi.SDPar_SetValue( hParMultipleIds, &saveMultipleIDs, sizeof(saveMultipleIDs) );

    // get rid of this word
    sdapi.SDWord_Delete( hSymphonVoc, hTempWord );

#ifdef DEBUG
    fclose( logFile );
#endif

    return hNewWord;
}

// ----------------------------------------------------------------------
// Break the spelling into spelling segments and get all pron segments for each
// each spelling segment. Put this info in an array SymbolChart
void PronHypo::createSymbolChart()
{
    char myWord[500];

    strcpy( myWord, spellingData );

    char* word = myWord;

    int maxLen = 0;

    Symbol* maxSym = 0;

    while( *word )
    {
        for( Symbol* sym = symStats->findFirstSymbol( word );
             sym;
             sym = symStats->findNextSymbol( word, sym ) )
        {
           int symsplen = strlen( sym->spellingString );
```

```
            if( strncmp( sym->spellingString, word, symsplen ) )
                continue;

            if( symsplen > maxLen )
            {
                maxLen = symsplen;
                maxSym = sym;
            }
        }

        // no longest symbol was found, like with underscore bug
        if( !maxSym )
        {
            word++; // do increment the position in the word, so that we
                    // won't get stuck at this position
            continue;
        }

        word += maxLen;

        // reset maxLen for next symbol
        maxLen = 0;
        // store symbol
        symbolChart[nSymbols] = (Symbol* )maxSym;
        // increment counter
        nSymbols++;
        // set back to NULL for next iteration
        maxSym = 0;
    }
;

//  ---------------------------------------------------------------------------
//
BOOL PronHypo::spellingDataIsDone()
{
    if( isDigitString ) return( *spellingData == '\0' );

    else
        return( ithSymbol == nSymbols );
}
```

```
//---------------------------------------------------------------------------
// expand to post processing step
void SymbolStatistics::postProcessInternalPron( unsigned char* sin,
                                                unsigned char* sout )

{
    switch( lang ) {
     case ENU:
        while( *sin != '\0' )
        {
            // cases like new
            if( *sin == 'M' )
            (
                *sout++ = 'P';
                *sout++ = 'y';
                sin++;
            }

            else
                *sout++ =  *sin++ ;
        )

        *sout = '\0';
        break;

     default:
        strcpy( (char*) sout, (char*) sin );
        break;
    )
)

// ---------------------------------------------------------------------------
// create prons with stress
void SymbolStatistics::addPronsWithStress( unsigned char* pronVars,
                                           unsigned char *arubaPron,
                                           int nVowels )

{
    unsigned char localPron[WORDLENGTH];

    int index;            // index needed to access the phoneme tables

    int count = 0;   // keeps track which phoneme has been dealt with

    int j  = 0;
int numVowel; // keeps track which phoneme needs to be changed

strcpy( (char*) localPron, (char*) arubaPron );

int pronLen = strlen( (char*) arubaPron );

localPron[ pronLen ] = 0;

for( int i = 0; i < nVowels; i++ )
{
    j = 0;

    // initialize
    numVowel = -1;

    for( unsigned char *s = localPron; *s; s++ )
    {
        // put pron in buffer
        pronVars[ i*WORDLENGTH + j ] = *s;

        // check which vowel of pron we are dealing with: i == count
        if( ( ( index = phonemeSet->isUnstressedVowel( *s ) ) != -1 )
            && i == count )
        {
            // number of vowels seen so far
            numVowel++;

            // is this phoneme the one we need to change
            if( numVowel != count )
            {
                j++;        // move index in pron array
                continue;
            }

            pronVars[ i*WORDLENGTH + j ] =
            phonemeSet->getVowelWtStress( index );

            int offset = ( i*WORDLENGTH ) + j+1;

            // copy rest of pron
            strcpy( (char*) pronVars+offset, (char*) s+1 );

            pronVars[ i*WORDLENGTH + pronLen ] = '\0';
```

```
                    count++;

                    break;
                )

                // counter in localPron
                j++;
            )
        }
)

// -------------------------------------------------------------------------
// extract pronunciation from linked list of hypos
void SymbolStatistics::getPronunciation( PronHypo *hypo,
                                         unsigned char* pronunciation,
                                         int doAcronym )

(
    int j = 0;
    int pronLength = 0;

    // post process pronunciation
    unsigned char arubaPron[WORDLENGTH];

    while( hypo )
    (
        int len = strlen( hypo->pronunciationString );

        if( ( pronLength + len) >= WORDLENGTH )
            // avoid overflow
            break;

        pronLength+=len;

        // verify whether it matches pr , the original pron
        while( len-- )
        (
            // '0' is the empty phoneme
            if( hypo->pronunciationString[ len ]  == '0' )
                continue;

            // no double phonemes
            if( (lang == ENU || lang == ENE) &&
                !doAcronym )
                if( arubaPron[ j-1 ] == hypo->pronunciationString[ len ] )
                    continue;

            arubaPron[j++] = hypo->pronunciationString[ len ];
        }

        hypo = hypo->next();
    }

    arubaPron[j]  = '\0';

    postProcessInternalPron( arubaPron, pronunciation );

    strrev( (char*) pronunciation );
}


// ---------------------------------------------------------------------------
// is word written in All capitalized letters : acronym
// get rid of '.', like  in I.B.M.
int SymbolStatistics::makeCap( LPCSTR str, LPSTR newStr )
{
    int isNotAllLower = 0;

    while( *str )
    {
        if( *str == '.' )
            str++;

        else if( *str == ' ' )
            str++;

        else if( isupper( *str ) )
        {
            *newStr++ = *str++;
            isNotAllLower = 1;
        }
        else
            *newStr++ = toupper(*str++);
    }

    *newStr = '\0';
    return isNotAllLower;      // all CAPS
}


//-----------------------------------------------------------------------
// returns number of new prons added
// pronunciation usually has no stress levels
// if a phone is stressed, the pron is added without further processing
void SymbolStatistics::addPronToState( LPCSTR spelling,
                                       unsigned char* pronunciation,
                                       int hypoScore, int mapStrategy )

{
    // this is possible for punctuation symbols
    if( *pronunciation == '\0' ) return;

    int j = 0;
    char modifiedPrompt[WORDLENGTH];
    SD_WORD temp = 0;

    // count of words added to voc
    numNewProns++;
    numAllNewProns++;

    // add a word with pronunciation
    sprintf( modifiedPrompt, "%s__%d", spelling, numNewProns );

    // check pronnciation: SAFETY (data files might have errors)
    if( !phonemeSet->isLegal( pronunciation ) )
    {
        char buf[256];
        wsprintf( buf, "\tERROR bad pronunciation: %s\n", pronunciation );
        OutputDebugString( buf );
        return;
    }

    temp = sdapi.SDWord_New( hSymphonVoc, modifiedPrompt );
    SDWord_SetPronunciationWithType( hSymphonVoc, temp, pronunciation,
                                     SD_PRONTYPE_GENERAL );
    sdapi.SDWord_SetLmlCount( hSymphonVoc, temp, 1 );
    SD_STATE hState = sdapi.SDState_GetHandle(hSymphonVoc, "temp", 0 );
    sdapi.SDState_AddWord(hSymphonVoc, hState, temp);

    hypoScore; // beats warning
```

```
#ifdef STRESSADDED
    // if there are already stressed syllables return
    if( phonemeSet->getNSylStressed( pronunciation ) != 0 )
        return;

    // determine number of unstressed vowels
    int nVowels = phonemeSet->getNumVowels( pronunciation );

    // the consonantal phoneme y can be hypothesized for i
    if( nVowels == 0 )
    (
        numNewPronsWithStress++;
        return;
    )

    // allocate space for stress pronunciations
    unsigned char* pronVars =  new unsigned char[ WORDLENGTH*nVowels ];
    // generate stress pronunciations: new prons are written into pronVars!!
    addPronsWithStress( pronVars, pronunciation, nVowels );

    if( numNewProns < 1000 )
        scoreArray[ temp%1000 ] = hypoScore;
    else
        assert("SYMPHON.CPP: more than 1000 word ids");

    // when not a digit string, adds stressed pronunciations
    for( int k = 0; k < nVowels; k++ )
    (
        numNewProns++;
        numAllNewProns++;

        // overWrite original pron
        strcpy( (char*) pronunciation, (char*) pronVars + ( k*WORDLENGTH ) );

        // check pronunciations
        if( !phonemeSet->isLegal( pronunciation ) )
            return;

        // add word to state
        temp = sdapi.SDWord_New( hSymphonVoc, modifiedPrompt );
        // TODO: make sdapi function
        SDWord_SetPronunciationWithType( hSymphonVoc, temp, pronunciation,
                                         SD_PRONTYPE_GENERAL );
        sdapi.SDWord_SetLm1Count( hSymphonVoc, temp, 1 );
        SD_STATE hState = sdapi.SDState_GetHandle(hSymphonVoc, "temp", 0 );
        sdapi.SDState_AddWord(hSymphonVoc, hState, temp);
        scoreArray[temp%1000] = hypoScore;
    }

    delete pronVars;
#endif

    };      mapStrategy;

}


// -----------------------------------------------------------------------------
//
void SymbolStatistics::cleanup()
{
    SD_WORD hWord;

    tempState = sdapi.SDState_GetHandle (hSymphonVoc, "temp", 0 );

    if( tempState )
    {
        SD_STATE_WORD_ITERATOR it;

        sdapi.SDState_IterateWords( hSymphonVoc, tempState, &it );

        while( ( hWord=sdapi.SDState_NextWord( &it ) ) != 0 )
        {
            // remove reference from state
            sdapi.SDState_DeleteWord( hSymphonVoc, tempState, hWord );
            //remove word from vocabulary
            sdapi.SDWord_Delete( hSymphonVoc, hWord );
        }

        sdapi.SDState_Delete (hSymphonVoc, tempState);
        tempState = 0;

    }

    // delete any garbage word that might be in the root
    SD_WORD_ITERATOR sdWIter;
    sdapi.SDWord_Iterate( hSymphonVoc, &sdWIter );
    while( ( hWord=sdapi.SDWord_Next( &sdWIter ) ) != 0 )
    {
        sdapi.SDWord_Delete( hSymphonVoc, hWord );
    }

    // clean up newWord structure and the idArray
    for( int i = 0; i < NUMSELECTED; i++)
        idArray[i] = 0;

    char vocName[WORDLENGTH];

    if( numAllNewProns > 30000 )
    {
        // TODO sdapi call
        SDVoc_GetFileName( hSymphonVoc, vocName, WORDLENGTH );

        sdapi.SDVoc_Close( hSymphonVoc );

        numAllNewProns = 0;
    }
}

// -------------------------------------------------------------------------
//
void SymbolStatistics::finish()
{
    if( hVocSoundAlike )
        sdapi.SDVoc_Close( hVocSoundAlike );

    cleanup();

    // cleanup score array.
    delete [] scoreArray;

    scoreArray = NULL;
}


// -------------------------------------------------------------------------
void SymbolStatistics::initializeState()
{
    // cleanup temp state
    if( InitTempState )
    {
        // initialize
        numNewProns = 0;
        numNewPronsWithStress = 0;
        cleanup();
    }

    // let's use pure SDAPI calls, will make it easier to go to server wrapper
    tempState = sdapi.SDState_New( hSymphonVoc, 0 );
    sdapi.SDState_SetName( hSymphonVoc, tempState, "temp" );

    assert( tempState );
}


// ---------------------------------------------------------------------
void SymbolStatistics::addToTablesIfNecessary( unsigned char* oemSpelling,
                                               unsigned char* pronunciation,
                                               int penalty )
{
    char *pr, *sp;

    unsigned char spelling[WORDLENGTH];

    OemToAnsi( (char*) oemSpelling, (char*)spelling );

    // the hoch gestellte drei is displayed as character 252, and gets
    // translated to an n in ansi. We want the hoghgestellte drei
    if( *oemSpelling == 0x00FC )
        *spelling = 0x00B3;


    for( Symbol* sym = s2pTable[ *spelling ]; sym; sym = (Symbol*) sym->next() )
    {
        if( sym->spellingString )
            if( 0 == strcmp( (char*)spelling, sym->spellingString ) )
                break;
    }

    if( sym )
    {
        sp = sym->spellingString;
    for( Phone* pho = sym->phone; pho; pho = (Phone*) pho->next() )
    {
        if( 0 == strcmp( (char *) pronunciation, (char * )pho->pronunciationString ) )
            break;
    }

    if( pho )
        return;
}
else
{
    // no Symbol exists, create one.
    sp = new char[ strlen( (char* ) spelling ) + 1 ];
    strcpy( sp, (char* ) spelling );

    sym = new Symbol( sp, 0 );
    sym->setNext( s2pTable[ *spelling ] );
    s2pTable[ *spelling ] = sym;
}

// if we got here, not in table

// see if the phone string exists anywhere
for( Phone* pho = p2sTable[ *pronunciation ]; pho; pho = (Phone*) pho->next() )
{
    if( 0 == strcmp( (char* )pronunciation, (char* ) pho->pronunciationString ) )
        break;
}

if( pho == 0 )
{
    // we are going to need the phone in the p2sTable... add it
    pr = new char[ strlen( (char *) pronunciation ) + 1 ];
    strcpy( (char* )pr, (char* )pronunciation );

    pho = new Phone( pr, penalty, strlen( (char*) pronunciation ) );
    pho->setNext( p2sTable[ *pronunciation ] );
    p2sTable[ *pronunciation ] = pho;
}
else
    pr = pho->pronunciationString;

Phone* newPho = new Phone( pr, penalty, strlen( (char*) pronunciation ) );
newPho->setNext( sym->phone );
```

```
    sym->phone = newPho;
    sym->nPhones++;

    // done with this pair ... go to next...
}

// ---------------------------------------------------------------------------
void SymbolStatistics::putInTables( unsigned char* spelling,
                                    unsigned char *pronunciation,
                                    char* pPenalty )

{
    unsigned char *prNext = pronunciation;
    unsigned char *spNext = spelling;

    int penalty = atoi( pPenalty );

    while( spNext && prNext )
    {
        spNext = (unsigned char *) parseToNextSpace( spelling = spNext );
        prNext = (unsigned char *) parseToNextSpace( pronunciation = prNext );

        addToTablesIfNecessary( spelling, pronunciation, penalty );
    }
}

// ---------------------------------------------------------------------------
// now the service functions:
Symbol* SymbolStatistics::sp2s( LPCSTR sp )
{
    assert( sp && *sp );

    for( Symbol* sym = s2pTable[ *sp ]; sym; sym = (Symbol*)sym->next() )
    {
        if( 0 == strcmp( sp, sym->spellingString ) )
            return sym;
    }
    return 0;
}

// ---------------------------------------------------------------------------
Phone* SymbolStatistics::pr2p( LPCSTR pr )
{
    for( Phone* pho = p2sTable[ *pr ]; pho; pho =(Phone*)pho->next() )
    {
        if( 0 == strcmp( pr, pho->pronunciationString ) )
            return pho;
    }
    return 0;
}


// ----------------------------------------------------------------------
Phone* SymbolStatistics::findPhone( LPCSTR sp, LPCSTR pr )
{
    Symbol *sym = sp2s( sp );

    if( sym == 0 )
        return 0;

    for( Phone *pho = sym->phone; pho; pho = (Phone*)pho->next() )
    {
        if( 0 == strcmp( pr, pho->pronunciationString ) )
            return pho;
    }
    return 0;
}

// ----------------------------------------------------------------------
Symbol* SymbolStatistics::findNextSymbol( LPCSTR sp, Symbol* sym )
{
    while( 0 != (sym=( Symbol* ) sym->next()) )
    {
        if( 0 == strncmp( sp, sym->spellingString, strlen( sym->spellingString ) ) )
            break;
    }
    return sym;
}


// ----------------------------------------------------------------------
// generates pronunciations

// pronunciation is used if a test wants to be done
// isdigitstring is an argument that will be used if we have a seperate
// way of generating digit prons.
void SymbolStatistics::spellingDecoder( PronHypoPQ* topPQ, LPCSTR spelling,
                                        unsigned char* pronunciation,
                                        int isDigitString )
{
    // get rid of warnings
    pronunciation;
    isDigitString;

    // initialize score array
    for( int n = 0; n < 1000; n++ )
        scoreArray[n] = 0;

    // initialize
    numNewProns = 0;

    PronHypoPQ hypoPQ( propagateCompareHypothesis, 15000 );

    // new Spelling has ck->K, x->xX, no punctuation and no digits
    PronHypo* hypo = new PronHypo( 0, 0, "", 0, 0, 0, spelling, this );

    symbolChart = new Symbol* [ WORDLENGTH ]; // that is at least

    // gives list of symbols for given spelling
    hypo->createSymbolChart();

    hypo->numOutputPhonemes = 0;

    hypo->isLetterString = 1;

    hypoPQ.push( hypo );

    int fullCnt = 0;

    while( 0 != (hypo=hypoPQ.pop()) )
    {
        if( hypo->spellingDataIsDone() )
        {
            topPQ->push( hypo );

            fullCnt = topPQ->count();

            if( fullCnt >= maxNumHypos )
            {
                // printf( "more than 300 hypos for: %s\n", spelling );
                delete [] symbolChart;
                return;
            }
        }
        else
        {
            if( hypo->propagateLongest( &hypoPQ ) == 0 )
                break;

            int cnt = hypoPQ.count();

            if( cnt > 14500 )
            {
                // we are generating too many hypos to be real time
                // so quit and work with what we have
                hypoPQ.removeEntriesTo( 0, cnt - 14000 );
                delete [] symbolChart;
                return;
            }
        }
    }

    delete [] symbolChart;
}
```